Gabriel Lucas Teles Vilaça

LAUNCHPAD F28027

Volume 1: Handles, Interrupções, PWM e ADC

Belo Horizonte - MG
2018

RESUMO

Este relatório foi feito com o intuito de auxiliar o entendimento da placa TMS320F28027, produzida pela *Texas Instruments* (TI), e que pertence à família C2000, que tem foco no controle, em tempo real, de malhas fechadas.

Este relatório será um ponto de partida para introduzir funções básicas disponíveis na *Launchpad* como: *Enhanced Pulse Width Modulator* (ePWM), *Analog to Digital Converter* (ADC), *Handles*, entre outros.

Cada capítulo contém um programa exemplo, que está comentado em português, para demonstrar como é feita a configuração de cada módulo e cada programa apresenta uma pequena utilidade para o módulo, como por exemplo, piscar um *Light Emitting Diode* (LED) ou modificar a intensidade luminosa do LED, utilizando o resultado de uma conversão do módulo ADC.

LISTA DE FIGURAS

LISTA DE SIGLAS E ABREVIATURAS

TI	*Texas Instruments*
DSP	*Digital Signal Processor*
ePWM	*Enhanced Pulse Width Modulator*
ADC	*Analog to Digital Converter*
LED	*Light Emitting Diode*
PWM	*Pulse Widht Modulation*
CPU	*Central Processing Unity*
GPIO	*General Purpose Input/Output*
I2C	*Inter-Integrade Circuit*
SPI	*Serial Peripheral Interface*
UART	*Universal Asynchronous Receiver-Transmitter*
JTAG	*Joint Test Action Group*
USB	*Universal Serial Bus*
CCS	*Code Composer Studio*
DRA	*Direct Register Access*
SD	*Software Driver*
PIE	*Peripheral Interrupt Expansion*
HSPCLK	*High Speed Peripheral Clock*
ADC	Conversor Analógico/Digital
SOC	*Start Of Conversion*

SUMÁRIO

1 INTRODUÇÃO

Os Microcontroladores e os *Digital Signal Processor* (DSP) apresentam várias características em comum. De fato, em relação à arquitetura, são extremamente semelhantes. As diferenças ficam evidentes quando se refere à capacidade de processamento de sinais. Os microcontroladores são circuitos digitais desenvolvidos para trabalhar tanto com manipulação de dados quanto para cálculos matemáticos. Por serem dispositivos genéricos, deve haver um compromisso entre essas duas aplicações, de forma que a estrutura projetada para atender estas funcionalidades não pode ser otimizada para ambas. Isto dificulta sua aplicação em sistemas em tempo real, onde a velocidade de cálculo é um parâmetro fundamental.

Os DSP, por outro lado, são dispositivos concebidos com foco no processamento. Seus circuitos são projetados de forma a aperfeiçoar o tempo de cálculo nas operações matemáticas, bem como realizá-las em um tempo previsível e contínuo, o que permite o processamento de grandes quantidades de dados. Em geral, as operações mais recorrentes, como adição e multiplicação, são realizadas antecipadamente e armazenados em acumuladores, de forma que um número maior de cálculos é feito em um único ciclo de processamento.

O kit de desenvolvimento *Piccolo Launchpad* é baseado no microcontrolador *TMS320F28027*, produzido pela TI. Ele pertence à família *Piccolo* da linha *C2000*, linha de microcontroladores para controle em tempo real. Não se trata, portanto, de um DSP, mas de um microcontrolador de baixo custo. Entretanto, ele nos oferece uma série de recursos que permitem realizar diversas aplicações em tempo real.

Neste trabalho, é desejado fornecer um ponto de partida para o desenvolvimento de aplicações no *Piccolo Launchpad*. Ele será estruturado como um tutorial. Os primeiros tópicos apresentam as características do F28027: O ambiente de desenvolvimento e os modelos de programação. Em seguida, serão abordados os periféricos ePWM e ADC*.* Explicando o funcionamento e dando exemplos de códigos comentados sobre como configurar esses módulos, de forma a facilitar o entendimento.

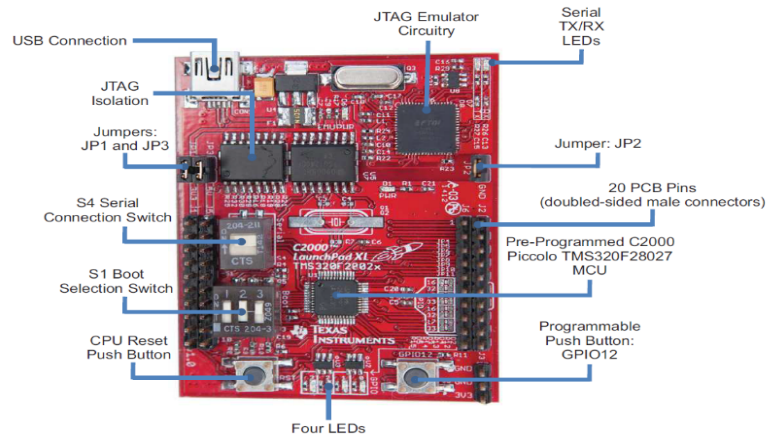
2 CARACTERÍSTICAS DO MICROCONTROLADOR

O microcontrolador TMS320F28027 apresenta as seguintes especificações:

- Arquitetura de 32 bits;
- *Clock* de 60MHz;
- ADC de 12 bits (16 canais multiplexados e 2 circuitos *Sample and Hold*);
- 8 módulos Pulse Width Modulation (PWM) com *timers* de 16 bits independentes;
- Módulo PWM de alta resolução;
- 3 *Central Processing Unity* (CPU) *Timers*;
- 22 pinos de General Purpose Input/Output (GPIO);
- Comunicação *Inter-Integrade Circuit* (I2C), *Serial Peripheral Interface* (SPI), *Universal Asynchronous Receiver-Transmitter* (UART);
- 64KB de memória *flash*;

O kit reúne *hardware* e *software* necessários para realizar aplicações baseadas no F28027 de maneira simples. A ferramenta *Joint Test Action Group* (JTAG) realiza a interface entre o microcontrolador e um computador pessoal para uma programação de fácil *debug* e testes. Além disso, ele possui uma interface de comunicação serial UART, que se comunica através de uma porta *Universal Serial Bus* (USB).

A placa apresenta quatro barramentos de dez pinos que se conectam aos pinos do microcontrolador, quatro *Light Emitting Diode* (LED) e duas chaves *push-bottom*. Isto facilita a interface entre o microcontrolador e o desenvolvedor na realização de testes. Serão utilizados alguns destes recursos quando os exemplos forem apresentados. A Figura 1 apresenta a placa Launchpad C2000.

Figura 1: Launchpad C2000.

Fonte:http://jaraidaniel.blogspot.com.br/2016/06/c2000-programming-basics-code-skeleton.html acessado em 28/03/18

2.1 Ambiente de desenvolvimento

O ambiente de desenvolvimento integrado recomendado pela TI é o Code Composer Studio (CCS). Neste trabalho, será utilizada a versão 6.1.3, disponibilizada, gratuitamente, no site da TI [1]. Entretanto, é necessário criar uma conta no site para fazer o *download*. O CCS foi desenvolvido pela própria TI e oferece diversas ferramentas para desenvolvimento e emulação em microcontroladores e DSP.

Juntamente com o CCS, será utilizado o software *ControlSuite*, que contém diversos *drivers*, manuais, bibliotecas e exemplos de muitos dispositivos fabricados pela Texas, incluindo documentação específica do *Piccolo Launchpad*, que é extremamente útil, principalmente aos iniciantes.

É necessário que o leitor instale os dois softwares, cujos links são mostrados na seção de Referências Bibliográficas, nos itens [1] e [2].

2.2 Modelos de programação

A TI disponibiliza no CCS os drivers necessários ao desenvolvimento no TMS320F28027. Ele oferece dois modelos de programação: *Direct Register Acces Model* (DRA) e *Software Driver Model* (SD). Os dois modelos podem ser utilizados independentemente ou combinados.

O modelo DRA trabalha em uma camada inferior ao SD, sendo necessário, portanto, um conhecimento sobre a arquitetura do dispositivo no nível de registradores. A linguagem de programação utilizada é a linguagem C. A vantagem desse modelo é a eficiência e o tamanho do código. Em aplicações que requerem um código mais otimizado, este modelo é o recomendável.

O modelo SD utiliza estruturas, chamadas de *handles* e tipos enumerados que abstraem o acesso aos registradores, tornando o desenvolvimento mais simples e códigos mais legíveis. É recomendado para aplicações que não necessitam de código e tempo de execução ótimos. Pode-se ainda utilizar ambos os modelos, caso a eficiência de código não seja um problema, mas o tempo de execução será um parâmetro crítico.

Ao longo do texto, será apresentado apenas o modelo de SD. O leitor que deseja saber mais sobre o modelo DRA poderá consultar a documentação de cada periférico, disponibilizada na seção de Referências Bibliográficas, no item [3]

3 HANDLES

Para configurar o DSP é necessário entender um recurso que é muito utilizado no modelo SD que são os *handles*. *Handle* é um termo em inglês que pode ser traduzido como: lidar com ou manusear.

3.1 Inicialização e definição

Para observar como os *handles* são utilizados será analisada, de forma superficial, uma parte do programa “Example_F2802xAdcSoc.C” que pode ser encontrado no *ControlSuite*. Ao abrir o arquivo, no CCS, o programa pode ser visto na tela do computador pessoal.

```
#include "DSP28x_Project.h"     // Device Headerfile and Examples Include File

#include "f2802x_common/include/adc.h"
#include "f2802x_common/include/clk.h"
#include "f2802x_common/include/flash.h"
#include "f2802x_common/include/gpio.h"
#include "f2802x_common/include/pie.h"
#include "f2802x_common/include/pll.h"
#include "f2802x_common/include/pwm.h"
#include "f2802x_common/include/wdog.h"

// Prototype statements for functions found within this file.
__interrupt void adc_isr(void);
void Adc_Config(void);

// Global variables used in this example:
uint16_t LoopCount;
uint16_t ConversionCount;
```

```
uint16_t Voltage1[10];
uint16_t Voltage2[10];

ADC_Handle myAdc;
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm;

void main(void)
{
   CPU_Handle myCpu;
   PLL_Handle myPll;
   WDOG_Handle myWDog;

   // Initialize all the handles needed for this application
   myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));
   myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
   myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
   myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
   myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
   myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
   myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
   myPwm = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
   myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));

   WDOG_disable(myWDog);
   CLK_enableAdcClock(myClk);
   (*Device_cal)();

   //Select the internal oscillator 1 as the clock source
```

```
CLK_setOscSrc(myClk, CLK_OscSrc_Internal);

// Setup the PLL for x10 /2 which will yield 50Mhz = 10Mhz * 10 / 2
PLL_setup(myPll, PLL_Multiplier_10, PLL_DivideSelect_ClkIn_by_2);
```

O primeiro detalhe que será analisado é que as variáveis: *myAdc, myClk, myCpu, myFlash, myGpio, myPie, myPll, myPwm, myWDog* são variáveis que possuem em seu nome um _Handle. Lembrando que *handle* pode ser traduzido como: lidar com ou manusear. A variável *myAdc* que é do tipo ADC_Handle pode ser interpretada como a variável que manuseia os registros do ADC do DSP.

Agora, clicando em ADC_Handle com o botão esquerdo do mouse enquanto se pressiona a tecla Ctrl, no CCS, o arquivo adc.h será aberto e nele é possível observar como é definido o ADC_Handle e o ADC_Obj, mostrado na Figura 2.

Figura 2: Definição do ADC_Handle e ADC_Obj

```
434 //! \brief Defines the analog-to-digital converter (ADC) object
435 //!
436 typedef struct _ADC_Obj_
437 {
438     volatile uint16_t      ADCRESULT[16];     //!< ADC result registers
439     volatile uint16_t      rsvd_1[26096];     //!< Reserved
440     volatile uint16_t      ADCCTL1;           //!< ADC Control Register 1
441     volatile uint16_t      rsvd_2[3];         //!< Reserved
442     volatile uint16_t      ADCINTFLG;         //!< ADC Interrupt Flag Register
443     volatile uint16_t      ADCINTFLGCLR;      //!< ADC Interrupt Flag Clear Register
444     volatile uint16_t      ADCINTOVF;         //!< ADC Interrupt Overflow Register
445     volatile uint16_t      ADCINTOVFCLR;      //!< ADC Interrupt Overflow Clear Register
446     volatile uint16_t      INTSELxNy[5];      //!< ADC Interrupt Select x and y Register
447     volatile uint16_t      rsvd_3[3];         //!< Reserved
448     volatile uint16_t      SOCPRICTRL;        //!< ADC Start Of Conversion Priority Control Register
449     volatile uint16_t      rsvd_4;            //!< Reserved
450     volatile uint16_t      ADCSAMPLEMODE;     //!< ADC Sample Mode Register
451     volatile uint16_t      rsvd_5;            //!< Reserved
452     volatile uint16_t      ADCINTSOCSEL1;     //!< ADC Interrupt Trigger SOC Select 1 Register
453     volatile uint16_t      ADCINTSOCSEL2;     //!< ADC Interrupt Trigger SOC Select 2 Register
454     volatile uint16_t      rsvd_6[2];         //!< Reserved
455     volatile uint16_t      ADCSOCFLG1;        //!< ADC SOC Flag 1 Register
456     volatile uint16_t      rsvd_7;            //!< Reserved
457     volatile uint16_t      ADCSOCFRC1;        //!< ADC SOC Force 1 Register
458     volatile uint16_t      rsvd_8;            //!< Reserved
459     volatile uint16_t      ADCSOCOVF1;        //!< ADC SOC Overflow 1 Register
460     volatile uint16_t      rsvd_9;            //!< Reserved
461     volatile uint16_t      ADCSOCOVFCLR1;     //!< ADC SOC Overflow Clear 1 Register
462     volatile uint16_t      rsvd_10;           //!< Reserved
463     volatile uint16_t      ADCSOCxCTL[16];    //!< ADC SOCx Control Registers
464     volatile uint16_t      rsvd_11[16];       //!< Reserved
465     volatile uint16_t      ADCREFTRIM;        //!< ADC Reference/Gain Trim Register
466     volatile uint16_t      ADCOFFTRIM;        //!< ADC Offset Trim Register
467     volatile uint16_t      rsvd_12[13];       //!< Reserved
468     volatile uint16_t      ADCREV;            //!< ADC Revision Register
469 } ADC_Obj;
470
471
472 //! \brief Defines the analog-to-digital converter (ADC) handle
473 //!
474 typedef struct ADC_Obj  *ADC_Handle;
```

Fonte: Acervo do aluno

Na última linha da Figura 2, observa-se que ADC_Handle é um ponteiro de um tipo de variável chamado ADC_Obj e logo acima se vê que ADC_Obj é um *typedef* que contem o tipo e o nome de cada registro do ADC.

No programa principal, como *myAdc* é uma variável do tipo ADC_Handle, que por sua vez é um ponteiro, logo *myAdc* também é um ponteiro, então a primeira coisa a ser feita na função *main* é inicializar todas variáveis que são derivadas de algum tipo que possui em seu nome _Handle, pois são ponteiros. Para isso é utilizada a função ADC_init((void*)ADC_BASE_ADDR,sizeof(ADC_Obj)) que foi projetada pela TI, essa função garante que *myAdc* será iniciado corretamente e que vai ser capaz de alterar os registros do ADC. Assim, *myAdc* foi inicializado com a posição correta e já pode ser utilizado como parâmetro nas funções.

3.2 Parâmetros

Na última linha do programa, tem-se a função WDOG_disable(*myWDog*) que utiliza como parâmetro o *myWDog,* que é uma variável do tipo WDOG_Handle, desta forma, a variável também possui todos os registros que estão relacionados ao *Watchdog*. A função WDOG_disable, como o próprio nome sugere, serve para desabilitar o *Watchdog* e ela utiliza o myWDog como parâmetro para obter acesso aos registros e então desabilitar de forma correta o *Watchdog*. Assim, evitando que o usuário gaste tempo procurando em manuais para encontrar em quais bits dos registradores do *Watchdog* é necessário colocar em nível alto ou baixo para desligar, facilitando a configuração dos periféricos para o usuário

4 INTERRUPÇÕES

Interrupções são eventos que fazem o processador desviar o programa de seu fluxo normal e então seguir para um trecho que é chamado de rotina de interrupção. Normalmente, as interrupções são processos de alta prioridade e não gastam muito tempo para execução, dessa forma não comprometem a execução do programa principal.

4.1 *Peripheral Interrupt Expansion*

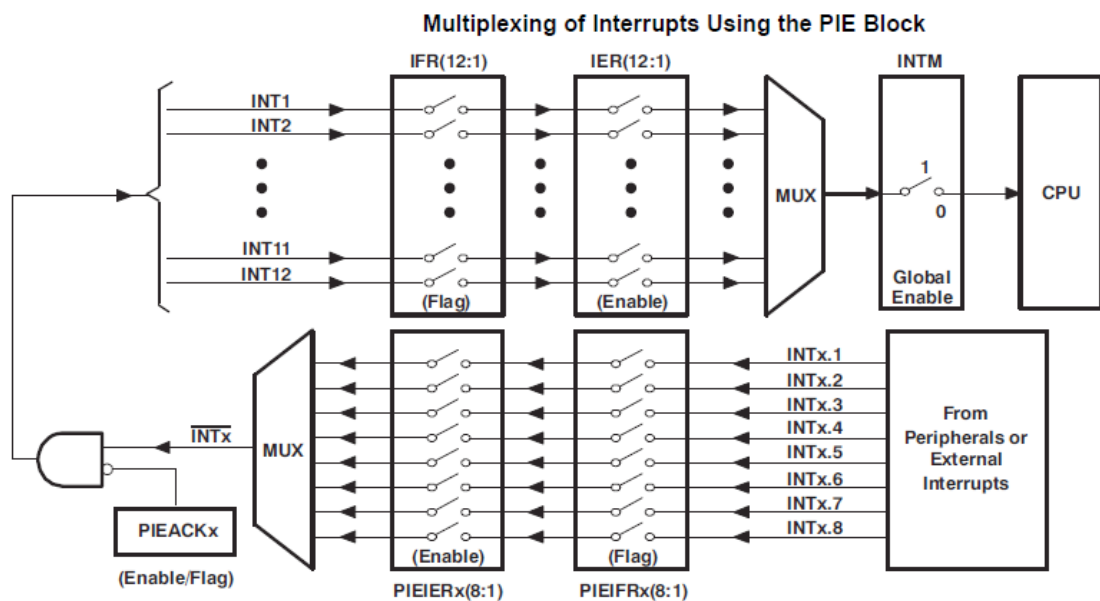
"A Peripheral Interrupt Expansion (PIE) é responsável por multiplexar conjuntos de interrupções em grupos menores. A PIE suporta 96 interrupções de fontes diferentes que são aglomeradas em grupos com 8 interrupções. Cada grupo envia um sinal em uma das doze linhas de INT1 até INT12." Traduzido de SPRUFN3D – [5]. A Figura 3 exemplifica essa descrição da PIE:

Figura 3: Multiplexação das interrupções na PIE

Multiplexing of Interrupts Using the PIE Block

IFR(12:1) IER(12:1) INTM

INT1
INT2
INT11
INT12
(Flag) (Enable)
MUX
1
0
Global Enable
CPU

INTx
MUX
PIEACKx
(Enable/Flag)
(Enable) (Flag)
PIEIERx(8:1) PIEIFRx(8:1)
INTx.1
INTx.2
INTx.3
INTx.4
INTx.5
INTx.6
INTx.7
INTx.8
From Peripherals or External Interrupts

Fonte: SPRUFN3D - [5]

A PIE será utilizada para habilitar as interrupções dos módulos: ADC e ePWM. Para isso, deve-se inicializar a tabela de vetores da PIE. A Figura 4 mostra uma parte da tabela, onde estão as interrupções do ADC e do ePWM que serão utilizadas. Para mais informações, esta tabela está disponível no SPRUFN3D - [5].

Figura 4: Tabela de interrupções da PIE

PIE Group 1 Vectors - MUXed into CPU INT1					
INT1.1	32	0x0000 0D40	2	ADCINT1	(ADC)
INT1.2	33	0x0000 0D42	2	ADCINT2	(ADC)
INT1.3	34	0x0000 0D44	2	Reserved	
INT1.4	35	0x0000 0D46	2	XINT1	
INT1.5	36	0x0000 0D48	2	XINT2	
INT1.6	37	0x0000 0D4A	2	ADCINT9	(ADC)
INT1.7	38	0x0000 0D4C	2	TINT0	(CPU-Timer0)
INT1.8	39	0x0000 0D4E	2	WAKEINT	(LPM/WD)
PIE Group 2 Vectors - MUXed into CPU INT2					
INT2.1	40	0x0000 0D50	2	EPWM1_TZINT	(EPWM1)
INT2.2	41	0x0000 0D52	2	EPWM2_TZINT	(EPWM2)
INT2.3	42	0x0000 0D54	2	EPWM3_TZINT	(EPWM3)
INT2.4	43	0x0000 0D56	2	EPWM4_TZINT	(EPWM4)
INT2.5	44	0x0000 0D58	2	Reserved	
INT2.6	45	0x0000 0D5A	2	Reserved	
INT2.7	46	0x0000 0D5C	2	Reserved	
INT2.8	47	0x0000 0D5E	2	Reserved	
PIE Group 3 Vectors - MUXed into CPU INT3					
INT3.1	48	0x0000 0D60	2	EPWM1_INT	(EPWM1)
INT3.2	49	0x0000 0D62	2	EPWM2_INT	(EPWM2)
INT3.3	50	0x0000 0D64	2	EPWM3_INT	(EPWM3)
INT3.4	51	0x0000 0D66	2	EPWM4_INT	(EPWM4)
INT3.5	52	0x0000 0D68	2	Reserved	
INT3.6	53	0x0000 0D6A	2	Reserved	
INT3.7	54	0x0000 0D6C	2	Reserved	
INT3.8	55	0x0000 0D6E	2	Reserved	-

Fonte: SPRUFN3D – [5]

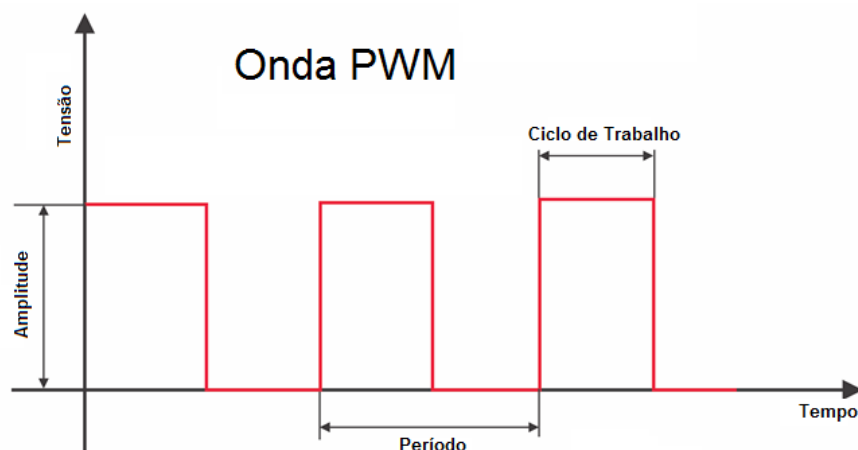
5 MÓDULO ePWM

O módulo ePWM pode gerar uma onda PWM. Sua função modular é o ciclo de trabalho em um canal de comunicação. Seu uso mais comum é controlar o valor de alimentação de uma carga.

O PWM consiste em comparar a saída de um contador, no caso do TMS320F28027, um contador de 16 bits que vai de 0x0000 até 0xFFFF. Também é possível alterar o valor máximo da contagem de 0xFFFF para um valor desejado pelo usuário.

5.1 Onda PWM

Antes de configurar o módulo ePWM, é preciso entender como é uma onda PWM e alguns termos relacionados a ela.

Figura 5: Onda PWM

Fonte: http://softwarelivre.blog.br/2014/07/21/gerando-pwm-com-raspberry-pi/, acessado em 23/03/18. Traduzido por GLT Vilaça

Analisando a Figura 5, é possível observar que a onda funciona em apenas dois valores de tensão, pode-se associar esses dois valores aos níveis lógicos, como em sistemas digitais, nível alto ou baixo. Período é o tempo gasto para completar um ciclo. Ciclo de trabalho é, em porcentagem, o quanto o sinal

permanece em nível lógico alto em relação ao período e pode ser calculado dividindo o tempo em que o sinal está em nível lógico alto pelo período.

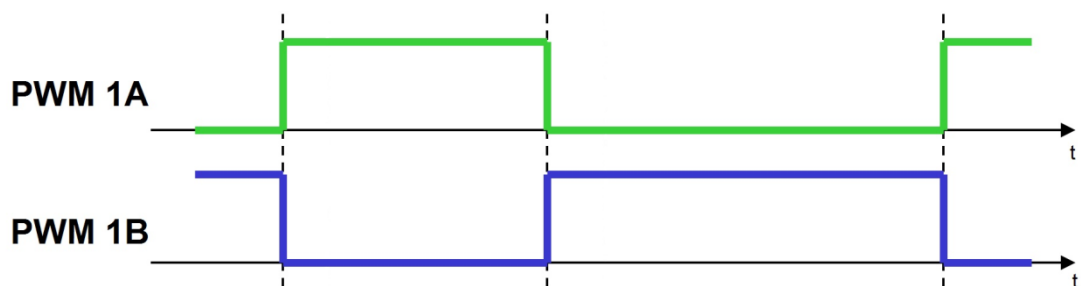
É possível criar um sinal PWM, chamado complementar. Um sinal PWM, em modo complementar, é quando existem dois sinais PWM de mesma frequência e fase, de tal forma que quando um dos sinais passa do nível lógico alto para o nível lógico baixo, o sinal que é complementar passa do nível lógico baixo para o nível lógico alto, assim um sinal está em nível lógico alto e outro nível em baixo, esse recurso é muito utilizado em conversores de energia. A Figura 6 mostra dois sinais PWM funcionando em modo complementar.

Figura 6: Onda PWM complementar

PWM 1A

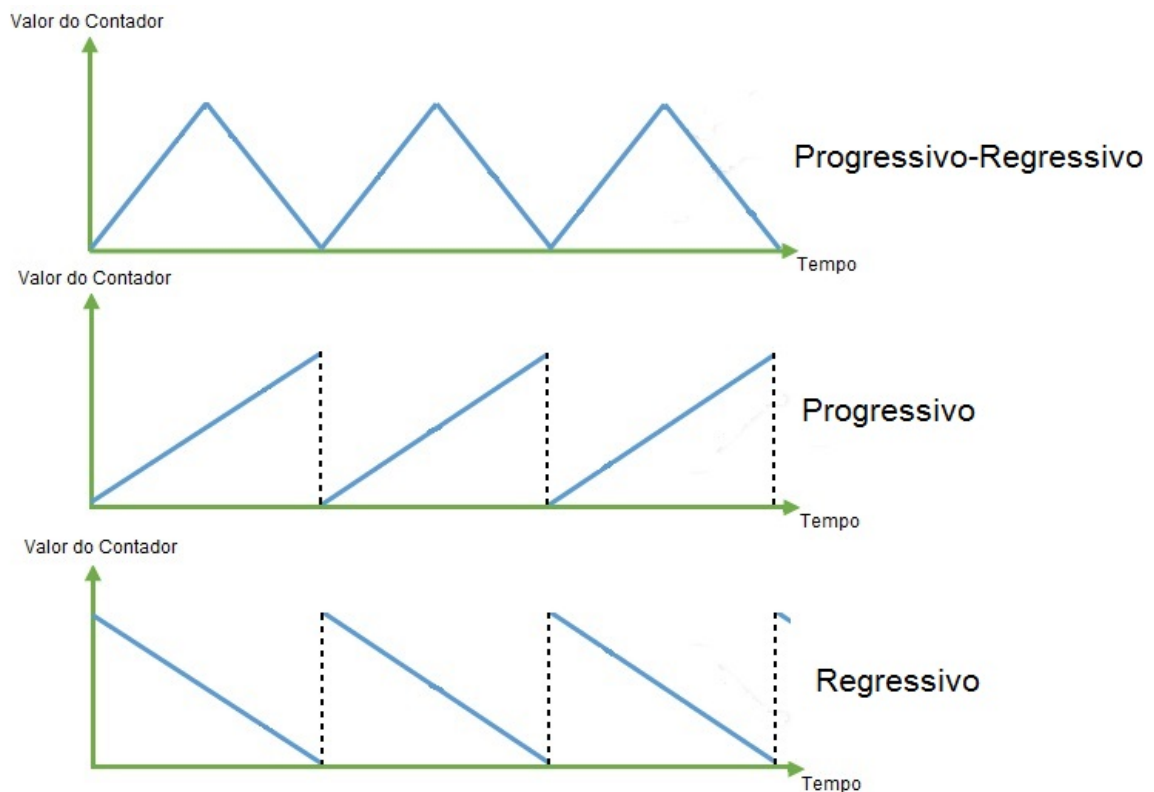
t

PWM 1B

t

Fonte: http://microchipdeveloper.com/pwr3101:pwm-edge-center-aligned-modes acessado 20/03/2018. Traduzido e editado por GLT Vilaça

5.2 Modos de Contagem e Clock

O módulo ePWM possui três modos de contagem que podem ser vistos na Figura 7:

Figura 7: Formas de contagem do módulo ePWM

Fonte: http://aimagin.com/blog/using-timercounter/, acessado 23/03/18. Editado por GLT Vilaça

O modo de contagem progressiva vai de 0x0000 até o valor máximo, que é definido pelo usuário, e reinicia seu ciclo de contagem. O modo de contagem regressiva é similar ao de progressiva, o contador começa do valor máximo, definido pelo usuário, e vai até 0x0000 e recomeça. No modo Progressivo-Regressivo, que é uma junção dos modos anteriores, é feita a contagem crescente de 0x0000 até o valor superior e depois de forma decrescente até 0x0000 e repete seu ciclo de contagem.

Em todos esses tipos de contagem, o usuário pode definir a ação de controle do sinal PWM gerado. Por exemplo: É possível fazer um sinal PWM no modo

Progressivo, em que o sinal fica em nível lógico baixo, quando o contador ultrapassar o valor a ser comparado e que alterna para nível lógico alto, quando seu valor for menor que o mesmo.

O *clock* que chega ao módulo ePWM é chamado de *High Speed Peripheral Clock* (HSPCLK) e o valor desse clock é a divisão do *clock* principal do sistema por: 1,2,4,6,8,10,12 ou 14. O valor do divisor é escolhido pelo programador e, por padrão, o valor do divisor é 2, após um *reset*.

Para calcular o valor do período da contagem, pode ser utilizada as seguintes fórmulas:

Para o modo Progressivo ou Regressivo:

$$\mathrm{T} = (Valor\ Máximo + 1) * HSPCLK$$

Para o modo Progressivo-Regressivo:

$$\mathrm{T} = 2 * Valor\ Máximo * HSPCLK$$

5.3 Programa 1

Este programa é feito para configurar o módulo ePWM, seu objetivo é gerar uma interrupção toda vez que o contador atingir o valor máximo, o estado do LED, ligado ao GPIO 2, será invertido, também gera um sinal PWM complementar nos terminais GPIO 0 e 1. A Figura 8 mostra a forma de onda do PWM complementar, vista no osciloscópio. A Figura 9 mostra o sinal da saída do GPIO 2. Este programa está disponível no Anexo 1.

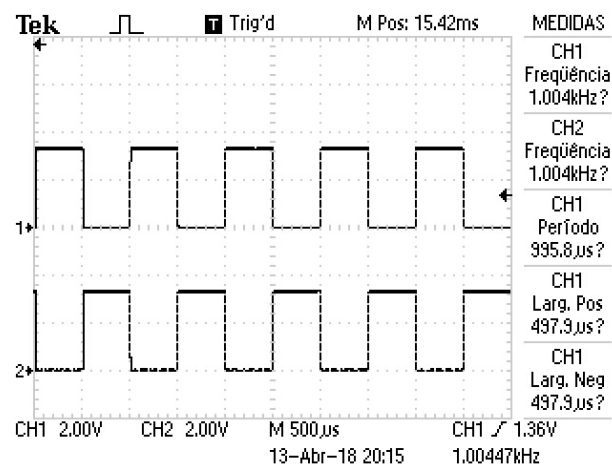
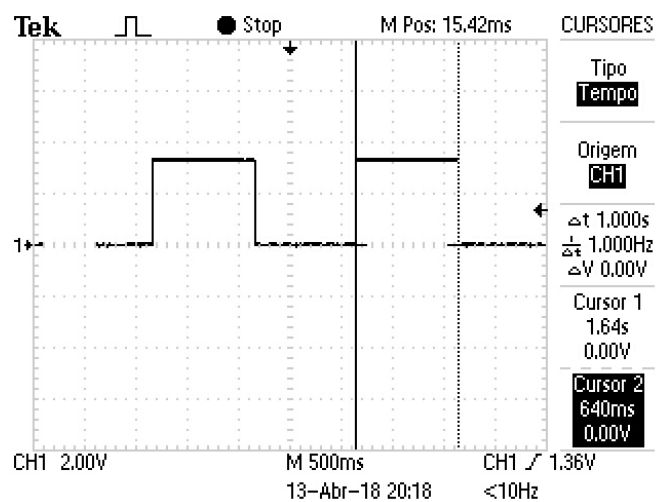
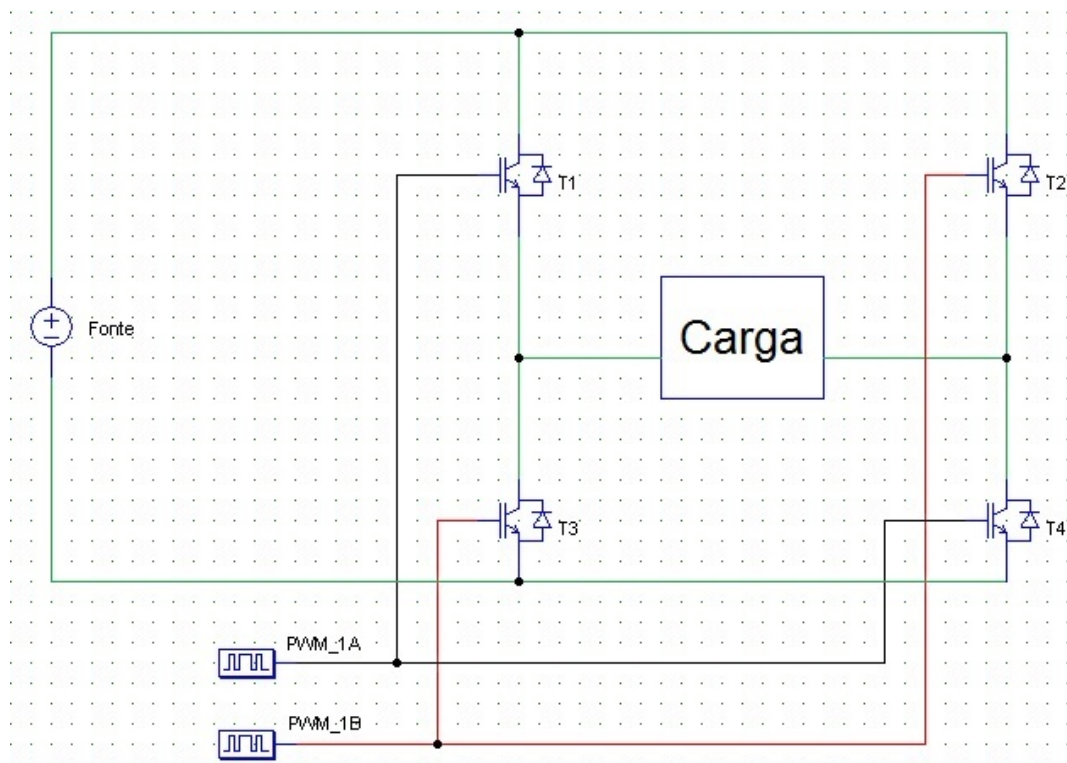
Figura 8: Sinal PWM complementar gerado

Fonte: Acervo do aluno

Figura 9: Sinal em GPIO2

Fonte: Acervo do aluno

5.4 Tempo morto

O tempo morto, em processamento de sinais, é definido como o tempo em que um processo permanece inativo.

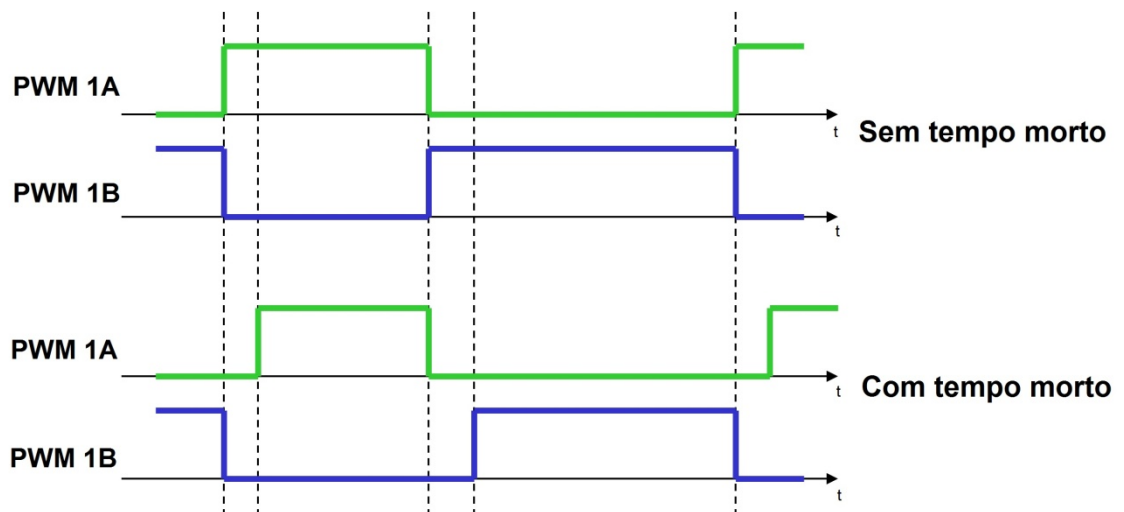
Para o módulo ePWM existe a possibilidade de configurar um tempo morto para o sinal PWM. Essa função é utilizada para controle de retificadores, em ponte, como visto na Figura 10.

Figura 10: Ponte H

Fonte: Acervo do aluno

No circuito da Figura 10, caso seja utilizado um sinal PWM, como aquele mostrado na Figura 6, para controlar os transistores, existirão momentos em que os transistores 1 e 3 ou 2 e 4 serão acionados simultaneamente, mesmo que por um curto período de tempo, dessa forma, ocorrerá um curto-circuito, podendo gerar danos aos transistores, ao filamento ou à fonte de energia. Dessa forma, é necessário que exista um tempo em que não haja sinal de ativação entre os

transistores, para evitar que isso ocorra, assim o sinal PWM, que controla os transistores, deve ser aquele da Figura 11, onde existe um espaço de tempo, onde os dois sinais não ativam transistores, assim contornando o problema. Esse espaço de tempo é denominado de tempo morto do sinal PWM.

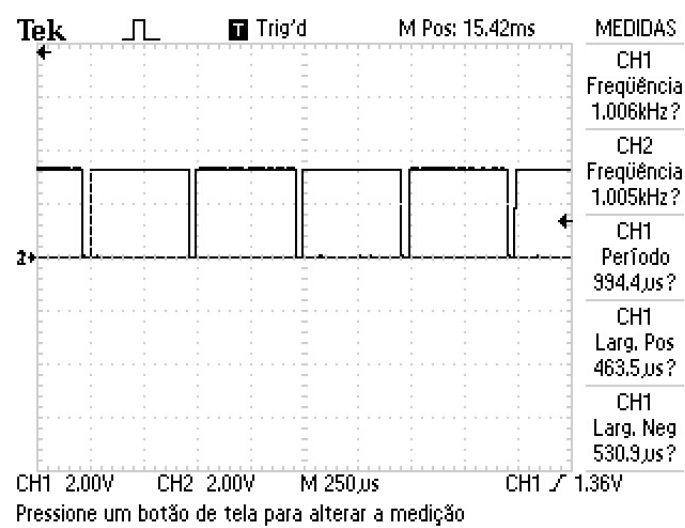
Figura 11: Onda PWM complementar com tempo morto

PWM 1A
PWM 1B
Sem tempo morto
PWM 1A
PWM 1B
Com tempo morto

Fonte: http://microchipdeveloper.com/pwr3101:pwm-edge-center-aligned-modes acessado 20/03/2018. Traduzido e editado por GLT Vilaça

5.5 Programa 2

Este programa é feito para configurar o módulo ePWM e girar um sinal PWM em modo complementar, onde existe tempo morto e a Figura 12 é a forma de onda do PWM dos sinais do GPIO 0 e 1, em modo complementar, que foram sobrepostas para ser possível visualizar o tempo morto entre os dois, no osciloscópio. Este programa está disponível no Anexo 2

Figura 12: Sinal PWM complementar com tempo morto gerado

Tek | Trig'd | M Pos: 15.42ms | MEDIDAS
CH1 Freqüência 1.006kHz?
CH2 Freqüência 1.005kHz?
CH1 Período 994.4µs?
CH1 Larg. Pos 463.5µs?
CH1 Larg. Neg 530.9µs?
CH1 2.00V CH2 2.00V M 250µs CH1 ⁄ 1.36V
Pressione um botão de tela para alterar a medição

Fonte: Acervo do aluno

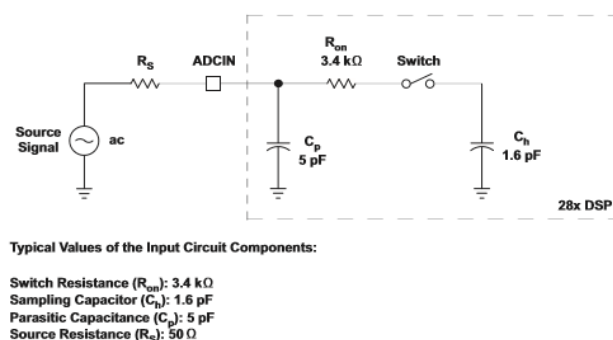
6 CONVERSOR ANALÓGICO PARA DIGITAL

Grande parte das grandezas controladas ou monitoradas por sistemas de controle são analógicas, ao contrário das variáveis digitais que apresentam um número finito de possibilidades que se podem encontrar, como exemplo, pode-se citar um botão que pode estar apertado ou não. As grandezas analógicas podem assumir infinitos valores a serem medidos como, por exemplo, a temperatura que pode assumir infinitos valores mesmo dentro de uma faixa de medição limitada.

Para usar esses valores analógicos em sistemas digitais é preciso utilizar um Conversor Analógico/Digital (ADC). A placa já tem com um ADC integrado e com a possibilidade de escolher entre os 16 canais multiplexados.

6.1 Amostragem do Sinal

Um circuito *sample and hold* é um circuito RC que é responsável por armazenar um valor de tensão no capacitor, para que possa ser convertido em um número digital, utilizando um conversor ADC. A Figura 13 mostra o circuito de *sample and hold* do ADC do TMS320F28027.

Figura 13: Circuito Sample and Hold

Fonte: SPRUGE5F - [4]

O programador pode escolher o tempo de amostragem, em ciclos de *clock*, gasto para fazer a medida do sinal.

6.2 Start of Conversion

Para que o ADC comece a converter um valor de tensão armazenado no circuito *sample and hold*, é necessário que haja um gatilho, que é o Start Of Conversion (SOC). Um SOC pode ser gerado por *software*, por um temporizador do DSP, ou mesmo, utilizando o módulo ePWM.

Como a melhor escolha para uma malha de controle é gerar conversões em intervalos de tempo fixos, será utilizado o módulo ePWM para gerar o SOC que irá iniciar a conversão do ADC, assim se obtêm os valores das grandezas monitoradas.

6.3 Programa 3

Este programa é feito para configurar o módulo ePWM e gerar um sinal PWM, em modo complementar, com tempo morto e gera um SOC, toda vez que o contador atingir o valor máximo da contagem, compara os valores de tensão medidos e liga o LED em GPIO3, dependendo dos valores lidos. Este programa está disponível no Anexo 3

7 REFERÊNCIAS BIBLIOGRÁFICAS

[1] Download do CCS, disponível em: <http://www.ti.com/tool/ccstudio>. Acessado em: 09 de julho de 2018

[2] Download do ControlSuite, disponível em: <http://www.ti.com/tool/controlsuite>. Acessado em: 09 de julho de 2018

[3] Texas Intruments. SPRUHX9: F2802x Peripheral Driver Library. Disponível em: <http://www.ti.com/lit/ug/spruhx9/spruhx9.pdf>. Acessado em: 09 de julho de 2018

[4] Texas Intruments. SPRUGE5F: Piccolo Analog-to-Digital Converter (ADC) and Comparator. Disponível em: <http://www.ti.com/lit/ug/spruge5f/spruge5f.pdf>. Acessado em: 09 de julho de 2018

[5] Texas Intruments. SPRUFN3D: Piccolo System Control and Interrupts. Disponível em: <http://www.ti.com/lit/ug/sprufn3d/sprufn3d.pdf>. Acessado em: 09 de julho de 2018

[6] Texas Intruments. SPRZ292M: Silicon Errata. Disponível em: <http://www.ti.com/lit/er/sprz292m/sprz292m.pdf>. Acessado em: 09 de julho de 2018

[7] Texas Intruments. SPRUGE9E: Enhanced Pulse Width Modulator (ePWM) Module. Disponível em: <http://www.ti.com/lit/ug/spruge9e/spruge9e.pdf >. Acessado em: 09 de julho de 2018

[8] Texas Intruments. SPRUGE8E: Piccolo High Resolution Pulse Width Modulator (HRPWM). Disponível em: <http://www.ti.com/lit/ug/spruge8e/spruge8e.pdf>. Acessado em: 09 de julho de 2018

[9] Texas Intruments. SPRU790D: Enhanced Quadrature Encoder Pulse (eQEP) Module. Disponível em: <http://www.ti.com/lit/ug/spru790d/spru790d.pdf>. Acessado em: 09 de julho de 2018

[10] Texas Intruments. SPRU566L: C2000 Real-Time Control Peripherals. Disponível em: <http://www.ti.com/lit/ug/spru566l/spru566l.pdf>. Acessado em: 09 de julho de 2018

[11] Texas Intruments. SPRUG71B: TMS320x2802x, 2803x Piccolo Serial Peripheral Interface (SPI).Disponível em: <http://www.ti.com/lit/ug/sprug71b/sprug71b.pdf >. Acessado em: 09 de julho de 2018

[12] Zappulla,Gustavo Sathler. Uma introdução ao C2000 lauchpad. Disponível em: <https://mega.nz/#!5AgQyCLY!p1Qywo9hYPb_TsUbAFHklnv7Wcau3-QU9a6ViiMWAlY >. Acessado em: 09 de julho de 2018

ANEXO 1

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo
#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog

// Declaração das funções que serão usadas mais a frente no programa
void Inicia_EPwm_Timer(void); // Declara a rotina de configuração da ePWM
void Inicia_Gpio(void); // Declara a rotina de configuração do GPIO
void Inicia_PIE(void); //Declara a rotina de configuração da PIE
__interrupt void epwm1_timer_isr(void); //Protótipo da rotina de interrupção

// Declara as variaveis que serão usadas ao longo do programa
unsigned int contador_de_interrupcoes=0; // Contador do número de interrupções

// Declara os handles globais
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm1;
ADC_Handle myAdc;
CPU_Handle myCpu;
```

```
int main(void)
	{
		// Declara handles locais
		PLL_Handle myPll;
		WDOG_Handle myWDog;

		// Inicializa os handles
		myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
		myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
		myFlash = FLASH_init((void *)FLASH_BASE_ADDR, sizeof(FLASH_Obj));
		myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
		myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
		myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
		myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR, sizeof(PWM_Obj));
		myWDog = WDOG_init((void *)WDOG_BASE_ADDR, sizeof(WDOG_Obj));
		myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));

		// Executa a inicialização básica do sistema:
		// Desabilita o Watchdog
		WDOG_disable(myWDog);
		// Habilita o clock do ADC
		CLK_enableAdcClock(myClk);
		(*Device_cal)();
		//Seleciona o oscilador interno 1 para ser o clock:
		CLK_setOscSrc(myClk, CLK_OscSrc_Internal);
		// Configura a PLL para x6 / 1 que resulta em 60Mhz = 10Mhz * 6 / 1:
		PLL_setup(myPll, PLL_Multiplier_6, PLL_DivideSelect_ClkIn_by_1);
```

```
// Desabilita a PIE e todas as interrupções, e limpa os flags para poder configurar os módulos:
PIE_disable(myPie);
PIE_disableAllInts(myPie);
CPU_disableGlobalInts(myCpu);
CPU_clearIntFlags(myCpu);

// [Usado no modo de configuração “FLASH” do CC] Copia funções da RAM para a RAM:
#ifdef _FLASH

// Copia as partes destinadas da memoria flash para a ram
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);

// Inicializa a memória flash
FLASH_setup(myFlash);
#endif

// Chamam as rotinas de configuração do PIE,ePWM e GPIO, respectivamente. Elas serão implementadas posteriormente
Inicia_PIE();
Inicia_EPwm_Timer();
Inicia_Gpio();

// Habilita as interrupções no sistema
PIE_enablePwmInt(myPie, PWM_Number_1);
CPU_enableInt(myCpu, CPU_IntNumber_3);
CPU_enableGlobalInts(myCpu);
CPU_enableDebugInt(myCpu);

for(;;); // Aguarda as interrupções
```

```
        }

        //Configura a PIE
void Inicia_PIE(void)
{
        // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
        PIE_setDebugIntVectorTable(myPie);
        PIE_enable(myPie);
        // Configura a rotina de interrupção do ePWM1 na PIE:(PIE Vector Table - [5])
                        PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_3,
PIE_SubGroupNumber_1,(intVec_t)&epwm1_timer_isr);
}
        // Rotina de configuração do ePWM
void Inicia_EPwm_Timer(void)
        {

    // Habilita o Clock no módulo para poder configura-lo
    CLK_disableTbClockSync(myClk);
    CLK_enablePwmClock(myClk, PWM_Number_1);

    //Faz o clk do PWM ser o 60M/20 = 30 MHz
    PWM_setHighSpeedClkDiv(myPwm1, PWM_HspClkDiv_by_1);
    PWM_setClkDiv(myPwm1, PWM_ClkDiv_by_2);

    // Liga a sincronia do PWM
    PWM_setSyncMode(myPwm1, PWM_SyncMode_EPWMxSYNC);

    // Permite a sincronia do PWM
    PWM_enableCounterLoad(myPwm1);

    // Define a fase do contador
    PWM_setPhase(myPwm1, 0);
```

```
// Define o valor máximo da contagem para 0x3A98
// assim o a frequencia do sinal PWM é 30M/(15000 * 2) = 1KHz e T = 0,001 s
PWM_setPeriod(myPwm1, 0x3A98);
// Coloca um valor no comparador A para gerar um ciclo de trabalho de 50%
PWM_setCmpA(myPwm1, 0x1D4C);
// Contagem Progressiva e Regressiva
PWM_setCounterMode(myPwm1, PWM_CounterMode_UpDown);
 // Gera uma interrupção toda vez que o contador chegar no valor maximo da
contagem
PWM_setIntMode(myPwm1, PWM_IntMode_CounterEqualPeriod);

 // Leva o nível lógico de EPWM1A a 1 quando, na contagem progressiva, o valor
de referência for igual ao contador

PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1,PWM_ActionQual_Set);
        // Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o
valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmA(myPwm1,PWM_ActionQual_Clear)
;

        // Leva o nível lógico de EPWM1B a 1 quando, na contagem progressiva, o
valor de referência for igual ao contador

PWM_setActionQual_CntUp_CmpA_PwmB(myPwm1,PWM_ActionQual_Clear);
        // Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o
valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmB(myPwm1,PWM_ActionQual_Set);

// Habilita Interrupção
```

```
PWM_enableInt(myPwm1);
// Gera interrupção em cada evento
PWM_setIntPeriod(myPwm1, PWM_IntPeriod_FirstEvent);

// Começa todos os PWM sincronizados
CLK_enableTbClockSync(myClk);

    }

    // Rotina que configura os GPIO 0, 1, 2
void Inicia_Gpio(void)
    {

    // Desabilita o Pull up nos pinos:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Disable);

    // Coloca o GPIO 0 para funcionar com o PWM1A:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
    // Coloca o GPIO 1 para funcionar com o PWM1B:
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);

    //Coloca o GPIO 2 no modo de proposito geral
    GPIO_setMode(myGpio,                          GPIO_Number_2,
GPIO_2_Mode_GeneralPurpose);

    // Define os pinos como saídas
    GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);
    GPIO_setDirection(myGpio, GPIO_Number_2, GPIO_Direction_Output);
```

```
        // Seta a saída em nível lógico baixo, o que apaga os LEDs:
        GPIO_setLow(myGpio, GPIO_Number_2);
        }

        // Rotinas de interrupção usadas neste exemplo
__interrupt void epwm1_timer_isr(void)
        {
                contador_de_interrupcoes++;

                //A cada 1000 interrupcoes o estado do led será alternado
                if(contador_de_interrupcoes==1000)
                {
                        GPIO_toggle(myGpio, GPIO_Number_2);
                        contador_de_interrupcoes=0;
                }
                // Limpa o flag de interrupção para poder receber novas interrupções
        PWM_clearIntFlag(myPwm1);
        PIE_clearInt(myPie, PIE_GroupNumber_3);
        }
```

ANEXO 2

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo
#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog

// Declaração das funções que serão usadas mais a frente no programa
void Inicia_EPwm_Timer(void); // Declara a rotina de configuração da ePWM
void Inicia_Gpio(void); // Declara a rotina de configuração do GPIO
void Inicia_PIE(void); //Declara a rotina de configuração da PIE
__interrupt void epwm1_timer_isr(void); //Protótipo da rotina de interrupção

// Declara as variaveis que serão usadas ao longo do programa
unsigned int contador_de_interrupcoes=0; // Contador do número de interrupções

// Declara os handles globais
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
PWM_Handle myPwm1;
ADC_Handle myAdc;
CPU_Handle myCpu;
```

```
int main(void)
	{
		// Declara handles locais
		PLL_Handle myPll;
		WDOG_Handle myWDog;

		// Inicializa os handles
		myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
		myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
		myFlash = FLASH_init((void *)FLASH_BASE_ADDR,
sizeof(FLASH_Obj));
		myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
		myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
		myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
		myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR,
sizeof(PWM_Obj));
		myWDog = WDOG_init((void *)WDOG_BASE_ADDR,
sizeof(WDOG_Obj));
		myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));

		// Executa a inicialização básica do sistema:
		// Desabilita o Watchdog
		WDOG_disable(myWDog);
		// Habilita o clock do ADC
		CLK_enableAdcClock(myClk);
		(*Device_cal)();
		//Seleciona o oscilador interno 1 para ser o clock:
		CLK_setOscSrc(myClk, CLK_OscSrc_Internal);
		// Configura a PLL para x6 / 1 que resulta em 60Mhz = 10Mhz * 6 / 1:
		PLL_setup(myPll, PLL_Multiplier_6, PLL_DivideSelect_ClkIn_by_1);
```

```
// Desabilita a PIE e todas as interrupções, e limpa os flags para poder configurar os módulos:
PIE_disable(myPie);
PIE_disableAllInts(myPie);
CPU_disableGlobalInts(myCpu);
CPU_clearIntFlags(myCpu);

// [Usado no modo de configuração “FLASH” do CC] Copia funções da RAM para a RAM:
#ifdef _FLASH

// Copia as partes destinadas da memoria flash para a ram
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);

// Inicializa a memória flash
FLASH_setup(myFlash);
#endif

// Chamam as rotinas de configuração do PIE,ePWM e GPIO, respectivamente. Elas serão implementadas posteriormente
Inicia_PIE();
Inicia_EPwm_Timer();
Inicia_Gpio();

// Habilita as interrupções no sistema
PIE_enablePwmInt(myPie, PWM_Number_1);
CPU_enableInt(myCpu, CPU_IntNumber_3);
CPU_enableGlobalInts(myCpu);
CPU_enableDebugInt(myCpu);

for(;;); // Aguarda as interrupções
```

```
        }

        //Configura a PIE
void Inicia_PIE(void)
{
        // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
        PIE_setDebugIntVectorTable(myPie);
        PIE_enable(myPie);
        // Configura a rotina de interrupção do ePWM1 na PIE:(PIE Vector Table - [5])
                        PIE_registerPieIntHandler(myPie,        PIE_GroupNumber_3,
PIE_SubGroupNumber_1,(intVec_t)&epwm1_timer_isr);
}
        // Rotina de configuração do ePWM
void Inicia_EPwm_Timer(void)
        {

    // Habilita o Clock no módulo para poder configura-lo
    CLK_disableTbClockSync(myClk);
    CLK_enablePwmClock(myClk, PWM_Number_1);

    //Faz o clk do PWM ser o 60M/20 = 30 MHz
    PWM_setHighSpeedClkDiv(myPwm1, PWM_HspClkDiv_by_1);
    PWM_setClkDiv(myPwm1, PWM_ClkDiv_by_2);

    // Liga a sincronia do PWM
    PWM_setSyncMode(myPwm1, PWM_SyncMode_EPWMxSYNC);

    // Permite a sincronia do PWM
    PWM_enableCounterLoad(myPwm1);

    // Define a fase do contador
    PWM_setPhase(myPwm1, 0);
```

```
// Define o valor máximo da contagem para 0x3A98
// assim o a frequencia do sinal PWM é 30M/(15000 * 2) = 1KHz e T = 0,001 s
PWM_setPeriod(myPwm1, 0x3A98);
// Coloca um valor no comparador A para gerar um ciclo de trabalho de 50%
PWM_setCmpA(myPwm1, 0x1D4C);
// Contagem Progressiva e Regressiva
PWM_setCounterMode(myPwm1, PWM_CounterMode_UpDown);
// Gera uma interrupção toda vez que o contador chegar no valor maximo da contagem
PWM_setIntMode(myPwm1, PWM_IntMode_CounterEqualPeriod);

// Leva o nível lógico de EPWM1A a 1 quando, na contagem progressiva, o valor de referência for igual ao contador

PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1,PWM_ActionQual_Set);
// Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmA(myPwm1,PWM_ActionQual_Clear);

// Leva o nível lógico de EPWM1B a 1 quando, na contagem progressiva, o valor de referência for igual ao contador

PWM_setActionQual_CntUp_CmpA_PwmB(myPwm1,PWM_ActionQual_Clear);
// Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmB(myPwm1,PWM_ActionQual_Set);
```

```
        //Tempo morto para sinais PWM em modo complementar ativos em nivel
lógico alto
    PWM_setDeadBandOutputMode(myPwm1,
PWM_DeadBandOutputMode_EPWMxA_Rising_EPWMxB_Falling);
    PWM_setDeadBandPolarity(myPwm1,
PWM_DeadBandPolarity_EPWMxB_Inverted);
    PWM_setDeadBandInputMode(myPwm1,
PWM_DeadBandInputMode_EPWMxA_Rising_and_Falling);
    PWM_setDeadBandRisingEdgeDelay(myPwm1, 0x03FF);
    PWM_setDeadBandFallingEdgeDelay(myPwm1, 0x03FF);

    // Habilita Interrupção
    PWM_enableInt(myPwm1);
    // Gera interrupção em cada evento
    PWM_setIntPeriod(myPwm1, PWM_IntPeriod_FirstEvent);

    // Começa todos os PWM sincronizados
    CLK_enableTbClockSync(myClk);

        }

        // Rotina que configura os GPIO 0, 1, 2
void Inicia_Gpio(void)
        {

        // Desabilita o Pull up nos pinos:
        GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Disable);
        GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Disable);
        GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Disable);

        // Coloca o GPIO 0 para funcionar com o PWM1A:
        GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
```

```
// Coloca o GPIO 1 para funcionar com o PWM1B:
GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);

//Coloca o GPIO 2 no modo de proposito geral
GPIO_setMode(myGpio, GPIO_Number_2,
GPIO_2_Mode_GeneralPurpose);

// Define os pinos como saídas
GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);
GPIO_setDirection(myGpio, GPIO_Number_2, GPIO_Direction_Output);

// Seta a saída em nível lógico baixo, o que apaga os LEDs:
GPIO_setLow(myGpio, GPIO_Number_2);
}

// Rotinas de interrupção usadas neste exemplo
__interrupt void epwm1_timer_isr(void)
{
    contador_de_interrupcoes++;

    //A cada 1000 interrupcoes o estado do led será alternado
    if(contador_de_interrupcoes==1000)
    {
        GPIO_toggle(myGpio, GPIO_Number_2);
        contador_de_interrupcoes=0;
    }
    // Limpa o flag de interrupção para poder receber novas interrupções
PWM_clearIntFlag(myPwm1);
PIE_clearInt(myPie, PIE_GroupNumber_3);
}
```

ANEXO 3

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "DSP28x_Project.h" // inclui biblioteca de arquivos do dispositivo
#include "f2802x_common/include/adc.h" // inclui biblioteca do ADC
#include "f2802x_common/include/clk.h" // inclui biblioteca do Clock
#include "f2802x_common/include/flash.h" // inclui biblioteca da memória FLASH
#include "f2802x_common/include/gpio.h" // inclui biblioteca do módulo GPIO
#include "f2802x_common/include/pie.h" // inclui biblioteca do módulo PIE
#include "f2802x_common/include/pll.h" // inclui biblioteca do módulo PLL
#include "f2802x_common/include/pwm.h" // inclui biblioteca dos módulos PWM
#include "f2802x_common/include/wdog.h" // inclui biblioteca do watchdog

// Declaração das funções que serão usadas mais a frente no programa
void Inicia_Adc(void); // Declara a rotina de configuração do ADC
void Inicia_EPwm_Timer(void); // Declara a rotina de configuração da ePWM
void Inicia_Gpio(void); // Declara a rotina de configuração do GPIO
void Inicia_PIE(void); //Declara a rotina de configuração da PIE
__interrupt void adc_isr(void); //Protótipo da rotina de interrupção
__interrupt void epwm1_timer_isr(void); //Protótipo da rotina de interrupção

// Declara as variaveis que serão usadas ao longo do programa
unsigned int contador_de_interrupcoes=0; // Contador do número de interrupções
unsigned int Tensao1 = 0; //Usada nas interrupções do ADC
unsigned int Tensao2 = 0; //Usada nas interrupções do ADC

// Declara os handles globais
CLK_Handle myClk;
FLASH_Handle myFlash;
GPIO_Handle myGpio;
PIE_Handle myPie;
```

```
PWM_Handle myPwm1;
ADC_Handle myAdc;
CPU_Handle myCpu;

int main(void)
	{
		// Declara handles locais
		PLL_Handle myPll;
		WDOG_Handle myWDog;

		// Inicializa os handles
		myClk = CLK_init((void *)CLK_BASE_ADDR, sizeof(CLK_Obj));
		myCpu = CPU_init((void *)NULL, sizeof(CPU_Obj));
		myFlash = FLASH_init((void *)FLASH_BASE_ADDR,
sizeof(FLASH_Obj));
		myGpio = GPIO_init((void *)GPIO_BASE_ADDR, sizeof(GPIO_Obj));
		myPie = PIE_init((void *)PIE_BASE_ADDR, sizeof(PIE_Obj));
		myPll = PLL_init((void *)PLL_BASE_ADDR, sizeof(PLL_Obj));
		myPwm1 = PWM_init((void *)PWM_ePWM1_BASE_ADDR,
sizeof(PWM_Obj));
		myWDog = WDOG_init((void *)WDOG_BASE_ADDR,
sizeof(WDOG_Obj));
		myAdc = ADC_init((void *)ADC_BASE_ADDR, sizeof(ADC_Obj));

		// Executa a inicialização básica do sistema:
		// Desabilita o Watchdog
		WDOG_disable(myWDog);
		// Habilita o clock do ADC
		CLK_enableAdcClock(myClk);
		(*Device_cal)();
		//Seleciona o oscilador interno 1 para ser o clock:
		CLK_setOscSrc(myClk, CLK_OscSrc_Internal);
```

```
// Configura a PLL para x6 / 1 que resulta em 60Mhz = 10Mhz * 6 / 1:
PLL_setup(myPll, PLL_Multiplier_6, PLL_DivideSelect_ClkIn_by_1);

// Desabilita a PIE e todas as interrupções, e limpa os flags para poder configurar os módulos:
PIE_disable(myPie);
PIE_disableAllInts(myPie);
CPU_disableGlobalInts(myCpu);
CPU_clearIntFlags(myCpu);

// [Usado no modo de configuração "FLASH" do CC] Copia funções da RAM para a RAM:
#ifdef _FLASH

// Copia as partes destinadas da memoria flash para a ram
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart, (size_t)&RamfuncsLoadSize);

// Inicializa a memória flash
FLASH_setup(myFlash);
#endif

// Chamam as rotinas de configuração do PIE,ePWM e GPIO, respectivamente. Elas serão implementadas posteriormente
Inicia_PIE();
Inicia_EPwm_Timer();
Inicia_Adc();
Inicia_Gpio();

// Habilita as interrupções no sistema
PIE_enablePwmInt(myPie, PWM_Number_1);
PIE_enableAdcInt(myPie, ADC_IntNumber_1);
```

```
            CPU_enableInt(myCpu, CPU_IntNumber_3);
            CPU_enableInt(myCpu, CPU_IntNumber_10);
                CPU_enableGlobalInts(myCpu);
                CPU_enableDebugInt(myCpu);

                for(;;); // Aguarda as interrupções
        }

        //Configura a PIE
void Inicia_PIE(void)
{
        // Configura a tabela de vetores do modo DEBUG e habilita a PIE:
        PIE_setDebugIntVectorTable(myPie);
        PIE_enable(myPie);
        // Configura a rotina de interrupção do ePWM1 na PIE:(PIE Vector Table - [5])
                        PIE_registerPieIntHandler(myPie,      PIE_GroupNumber_3,
PIE_SubGroupNumber_1,(intVec_t)&epwm1_timer_isr);
                        PIE_registerPieIntHandler(myPie,      PIE_GroupNumber_10,
PIE_SubGroupNumber_1,(intVec_t)&adc_isr);
}
        // Rotina de configuração do ePWM
void Inicia_EPwm_Timer(void)
{

    // Habilita o Clock no módulo para poder configura-lo
    CLK_disableTbClockSync(myClk);
    CLK_enablePwmClock(myClk, PWM_Number_1);

    //Faz o clk do PWM ser o 60M/20 = 30 MHz
    PWM_setHighSpeedClkDiv(myPwm1, PWM_HspClkDiv_by_1);
    PWM_setClkDiv(myPwm1, PWM_ClkDiv_by_2);
```

```
// Liga a sincronia do PWM
PWM_setSyncMode(myPwm1, PWM_SyncMode_EPWMxSYNC);

// Permite a sincronia do PWM
PWM_enableCounterLoad(myPwm1);

// Define a fase do contador
PWM_setPhase(myPwm1, 0);

// Define o valor máximo da contagem para 0x3A98
// assim o a frequencia do sinal PWM é 30M/(15000 * 2) = 1KHz e T = 0,001 s
PWM_setPeriod(myPwm1, 0x3A98);
// Coloca um valor no comparador A para gerar um ciclo de trabalho de 50%
PWM_setCmpA(myPwm1, 0x1D4C);
// Contagem Progressiva e Regressiva
PWM_setCounterMode(myPwm1, PWM_CounterMode_UpDown);
 // Gera uma interrupção toda vez que o contador chegar no valor maximo da
contagem
PWM_setIntMode(myPwm1, PWM_IntMode_CounterEqualPeriod);

 // Leva o nível lógico de EPWM1A a 1 quando, na contagem progressiva, o valor
de referência for igual ao contador

PWM_setActionQual_CntUp_CmpA_PwmA(myPwm1,PWM_ActionQual_Set);
    // Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o
valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmA(myPwm1,PWM_ActionQual_Clear)
;

    // Leva o nível lógico de EPWM1B a 1 quando, na contagem progressiva, o
valor de referência for igual ao contador
```

```
PWM_setActionQual_CntUp_CmpA_PwmB(myPwm1,PWM_ActionQual_Clear);
        // Leva o nível lógico de EPWM1A a 0 quando, na contagem regressiva, o valor de referência for igual ao contador

PWM_setActionQual_CntDown_CmpA_PwmB(myPwm1,PWM_ActionQual_Set);

        //Tempo morto para sinais PWM em modo complementar ativos em nivel lógico alto
PWM_setDeadBandOutputMode(myPwm1, PWM_DeadBandOutputMode_EPWMxA_Rising_EPWMxB_Falling);
PWM_setDeadBandPolarity(myPwm1, PWM_DeadBandPolarity_EPWMxB_Inverted);
PWM_setDeadBandInputMode(myPwm1, PWM_DeadBandInputMode_EPWMxA_Rising_and_Falling);
   PWM_setDeadBandRisingEdgeDelay(myPwm1, 0x03FF);
   PWM_setDeadBandFallingEdgeDelay(myPwm1, 0x03FF);

   // Habilita Interrupção
   PWM_enableInt(myPwm1);
   // Gera interrupção em cada evento
   PWM_setIntPeriod(myPwm1, PWM_IntPeriod_FirstEvent);

   //Configuração do SOC gerado pelo PWM
   // Habilita a geração de pulsos para SOCs (grupo A)
   PWM_enableSocAPulse(myPwm1);
    // Seleciona a condição para geração do pulso. Nesse caso, o disparo ocorrerá quando o valor do contador da PWM for igual ao período
   PWM_setSocAPulseSrc(myPwm1, PWM_SocPulseSrc_CounterEqualPeriod);
   // Define que o pulso será gerado todo evento
   PWM_setSocAPeriod(myPwm1, PWM_SocPeriod_FirstEvent);
```

```
    // Começa todos os PWM sincronizados
    CLK_enableTbClockSync(myClk);

}
void Inicia_Adc(void)
{
        //Inicialização basica do ADC
    ADC_enableBandGap(myAdc);
    ADC_enableRefBuffers(myAdc);
    ADC_powerUp(myAdc);
    ADC_enable(myAdc);
    ADC_setVoltRefSrc(myAdc, ADC_VoltageRefSrc_Int);

      //Garante que a interrupção só ocorre depois que todos os valores foram
convertidos
    ADC_setIntPulseGenMode(myAdc, ADC_IntPulseGenMode_Prior);
    //Habilita a interrupção 1 do ADC
    ADC_enableInt(myAdc, ADC_IntNumber_1);
    //Impede que nova interrupção seja gerada até que o flag seja resetado
    ADC_setIntMode(myAdc, ADC_IntNumber_1, ADC_IntMode_ClearFlag);

    //Faz com que a o Final da conversão do ADC 2 gere a interrupção
    ADC_setIntSrc(myAdc, ADC_IntNumber_1, ADC_IntSrc_EOC2);
    //Faz com que o PWM1 gere os SOC 0,1 e 2
                        ADC_setSocTrigSrc(myAdc,     ADC_SocNumber_0,
ADC_SocTrigSrc_EPWM1_ADCSOCA);
                        ADC_setSocTrigSrc(myAdc,     ADC_SocNumber_1,
ADC_SocTrigSrc_EPWM1_ADCSOCA);
                        ADC_setSocTrigSrc(myAdc,     ADC_SocNumber_2,
ADC_SocTrigSrc_EPWM1_ADCSOCA);
    //Faz com que as conversões ocorram em cascata
```

```
ADC_setSocChanNumber (myAdc, ADC_SocNumber_0, ADC_SocChanNumber_A4);
ADC_setSocChanNumber (myAdc, ADC_SocNumber_1, ADC_SocChanNumber_A4);
ADC_setSocChanNumber (myAdc, ADC_SocNumber_2, ADC_SocChanNumber_A2);
//Define o tempo de amostragem dos sinais como 7 ciclos de clock
ADC_setSocSampleWindow(myAdc, ADC_SocNumber_0, ADC_SocSampleWindow_7_cycles);
ADC_setSocSampleWindow(myAdc, ADC_SocNumber_1, ADC_SocSampleWindow_7_cycles);
ADC_setSocSampleWindow(myAdc, ADC_SocNumber_2, ADC_SocSampleWindow_7_cycles);
}

    // Rotina que configura os GPIO 0, 1, 2
void Inicia_Gpio(void)
{

    // Desabilita o Pull up nos pinos:
    GPIO_setPullUp(myGpio, GPIO_Number_0, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_1, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_2, GPIO_PullUp_Disable);
    GPIO_setPullUp(myGpio, GPIO_Number_3, GPIO_PullUp_Disable);

    // Coloca o GPIO 0 para funcionar com o PWM1A:
    GPIO_setMode(myGpio, GPIO_Number_0, GPIO_0_Mode_EPWM1A);
    // Coloca o GPIO 1 para funcionar com o PWM1B:
    GPIO_setMode(myGpio, GPIO_Number_1, GPIO_1_Mode_EPWM1B);

    //Coloca o GPIO 2 e 3 no modo de proposito geral
```

```
        GPIO_setMode(myGpio, GPIO_Number_2, GPIO_2_Mode_GeneralPurpose);
        GPIO_setMode(myGpio, GPIO_Number_3, GPIO_3_Mode_GeneralPurpose);

        // Define os pinos como saídas
        GPIO_setDirection(myGpio, GPIO_Number_0, GPIO_Direction_Output);
        GPIO_setDirection(myGpio, GPIO_Number_1, GPIO_Direction_Output);
        GPIO_setDirection(myGpio, GPIO_Number_2, GPIO_Direction_Output);
        GPIO_setDirection(myGpio, GPIO_Number_3, GPIO_Direction_Output);

        // Seta a saída em nível lógico baixo, o que apaga os LEDs:
        GPIO_setLow(myGpio, GPIO_Number_2);
        GPIO_setLow(myGpio, GPIO_Number_3);
}

        // Rotinas de interrupção usadas neste exemplo
__interrupt void epwm1_timer_isr(void)
{
                contador_de_interrupcoes++;

                //A cada 1000 interrupcoes o estado do led será alternado
                if(contador_de_interrupcoes==1000)
                {
                        GPIO_toggle(myGpio, GPIO_Number_2);
                        contador_de_interrupcoes=0;
                }
                // Limpa o flag de interrupção para poder receber novas interrupções
        PWM_clearIntFlag(myPwm1);
        PIE_clearInt(myPie, PIE_GroupNumber_3);
}
```

```
__interrupt void adc_isr(void)
{
	Tensao1 = ADC_readResult(myAdc, ADC_ResultNumber_1);
	Tensao2 = ADC_readResult(myAdc, ADC_ResultNumber_2);

	//Liga o LED se a tensão1 for maior que a tensão2 e apaga caso contrario
	if(Tensao1>Tensao2)
	{
		GPIO_setLow(myGpio, GPIO_Number_3);
	}
	else
	{
		GPIO_setHigh(myGpio, GPIO_Number_3);
	}
	// Limpa o flag de ADCINT1 para o próximo evento
	ADC_clearIntFlag(myAdc, ADC_IntNumber_1);
	// Limpa a interrupção no PIE
	PIE_clearInt(myPie, PIE_GroupNumber_10);
	}
```